CPSC 321 Computer Architecture
Fall Semester 2004
Pieplined MIPS–Lite Processor in Verilog
Due Date: Tuesday, 12/07/2004, at 23:59:59, via `turnin`.
Work in groups of up to two students

---

1 Objective

In this project you will pipeline the single-cycle processor you built in the previous project. The instruction set is exactly the same as the one in the previous project. The processor includes data-path and a control unit which can be similar to those discussed in class and in Chapter 6.

2 Pipelining Your MIPS-Lite Design

2.1 Pipelining

You have to implement the following five-stage pipeline:

IF Instruction Fetch: Access the instruction cache for the instruction to be executed.

ID Instruction Decode: Decode the instruction and read the operands from the register file. For branch instructions, calculate the branch target instruction address and compare the registers.

EX Execute: Bypass operands from other pipeline stages. One of the following activities can occur:

- For computational instructions, the ALU or the shifter performs the arithmetic or logical instructions.
- For load or store instructions, the data address is calculated.

MEM Data Memory Access: Access the data cache for load or store instructions.

WB Write Back: Write result to register file.

Add the appropriate pipeline registers to your single cycle design. Assume that memory accesses take only one cycle in your implementation.

2.2 Initial Testing

To test your program, you should write diagnostic programs, similar to those that you wrote in Project 2. The general structure of the programs should be some calculations, data manipulations, etc., followed by `sw`'s to the main memory. Then, at the end of the program, and at points during

the execution, you can dump memory to show that the correct values were correctly written to memory.

Keep in mind that you haven't handled hazards yet (Problem 3), so you must be careful that your test programs don't try to use values too soon after they are generated.

2.3 Pipelining Gain

Calculate the cycle time for your pipelined processor. In your writeup, compare the cycle time from Project 2 to that of Project 3.

3 Handling Pipeline Hazards

3.1 Handle Hazards

Here is a list of the hazards you must handle:

Data Hazards Data hazards occur when the data produced by an instruction is used as an operand by subsequent instructions.

- The ALU is one source of data hazards. Add the necessary forwarding logic and buses so that the result of an ALU operation can immediately be used by the following three instructions without waiting for the data to be written in the register file.
- Load instructions also present a data hazard because the data is not available until the end of the MEM phase. The MIPS instruction set specifies that load instructions have a one cycle delay. That means that the compiler cannot generate code sequences where the data of a load will be used during the following cycle. Implement your data-path so that it has one load delay slot.

Control Hazards Branch instructions present a common case of control hazards. Comply with the MIPS instruction set definition and implement your processor so that it has one branch delay slot that is always executed regardless of the result of the branch.

Structural hazards are there any in this design? If so, explain what they are and how you are handling them. If not, why not?

Write or modify programs to test all the different hazard cases. Remember that hazards do not necessarily occur between two adjacent instructions. They can happen between two instructions that are separated by another instruction (or two?). Consider the following lines of code:

```
add    $1, $2, $3
sll    $5, $6
sub    $6, $1, $7
```

The `add` and `sub` instructions have a data hazard, yet there is an `sll` between them. Be sure to check these kinds of cases.

3.2 Pipeline Interlocks/Interlocking Loads

Now that you have basic hazards dealt with, you should figure out how to handle pipeline stalls. Your current processor deals with the load delay slot in the same way as the original version of MIPS: If the compiler generates a code sequence in which a value is loaded from memory and used by the next instruction, the following instruction gets the wrong value. Of course, the instruction specification explicitly disallowed such code sequences; if no other options were available, the compiler would have to introduce a `noop` in the load delay slot to avoid getting the wrong answer.

As your final exercise, introduce a pipeline stall so that a value can be used by the compiler in the very next cycle after it is loaded from memory. This feature was added to later versions of the MIPS instructions set. To be clear, we want the following code sequence to do the "obvious" thing, i.e. the add should use the value loaded from memory:

```
lw      $1, 4($2)
add     $2, $1, $3
```

Make sure to rerun all of your tests from Part–III.a to verify that you haven't broken anything. Write tests that try several different distances between loads and their following values.

4 Deliverables

Design Notebook (30pts)
Please keep a design notebook that documents your design process and all the steps in the program development; it should document the progress that you made while writing your code (keep track of date and time). Write the design notebook while you are developing the software, do not write it after the fact.
Your notebook should keep a log of all the errors that you make; it should be handwritten with a pen that is not erasable; printouts documenting your progress should be glued in. The grading of your design notebook will take into account the clarity, regularity, legibility and organization of your annotations.
You also need to turnin the control matrix for the control unit.

Verilog Program (40pts)
Turn in your verilog program via the *turnin* command. Ensure that your program is clearly commented.

Project Demonstration (30pts)
Project demonstrations should be completed within a week after submission. Failure to demonstrate will result in a Fail grade on the project.

5 Dishonesty

Make sure that you complete the assignment by yourself. Do not copy the code from others, nor provide others with your code. Refrain from copying and modifying the code from other sources.